Universidad Andrés Bello

S8: Recursividad

Estructuras de Datos

Docente: Pamela Landero Sepúlveda
pamelalandero@gmail.com

Definición

- Consiste en definir valor de una función en términos de instancias más simples, menores o anteriores de la misma función, llegando hasta una instancia tan simple que el valor se determina en forma inmediata.

- Una función que se llama a sí misma, directa o indirectamente, es una **función recursiva**.

- La recursión es un potente concepto con el que se pueden expresar ciertos procedimientos de cálculo muy elegantemente.

- No obstante, al principio cuesta un poco entender las funciones recursivas y un poco más diseñar nuestras propias funciones recursivas.

- La recursión es un concepto difícil cuando se está aprendiendo a programar.

¿Cómo funciona?

- Cada llamada recursiva siempre debe tender a un *caso base* que puede resolverse sin recursión.
- El caso base puede no realizar acción alguna, pero lo importante es que se llegue a un punto en el que no se produzcan más auto-llamadas.
- Para alcanzar el caso base, es necesario que los parámetros de cada llamada recursiva, deben corresponder a un caso más pequeño del problema inicial.
- Tras el caso base, se producirá un retorno en el que el resultado de cada llamada se irá incorporando al de la llamada anterior hasta obtener un resultado definitivo que será devuelto a la sentencia que realizó la primera llamada al algoritmo.

¿Cómo funciona?

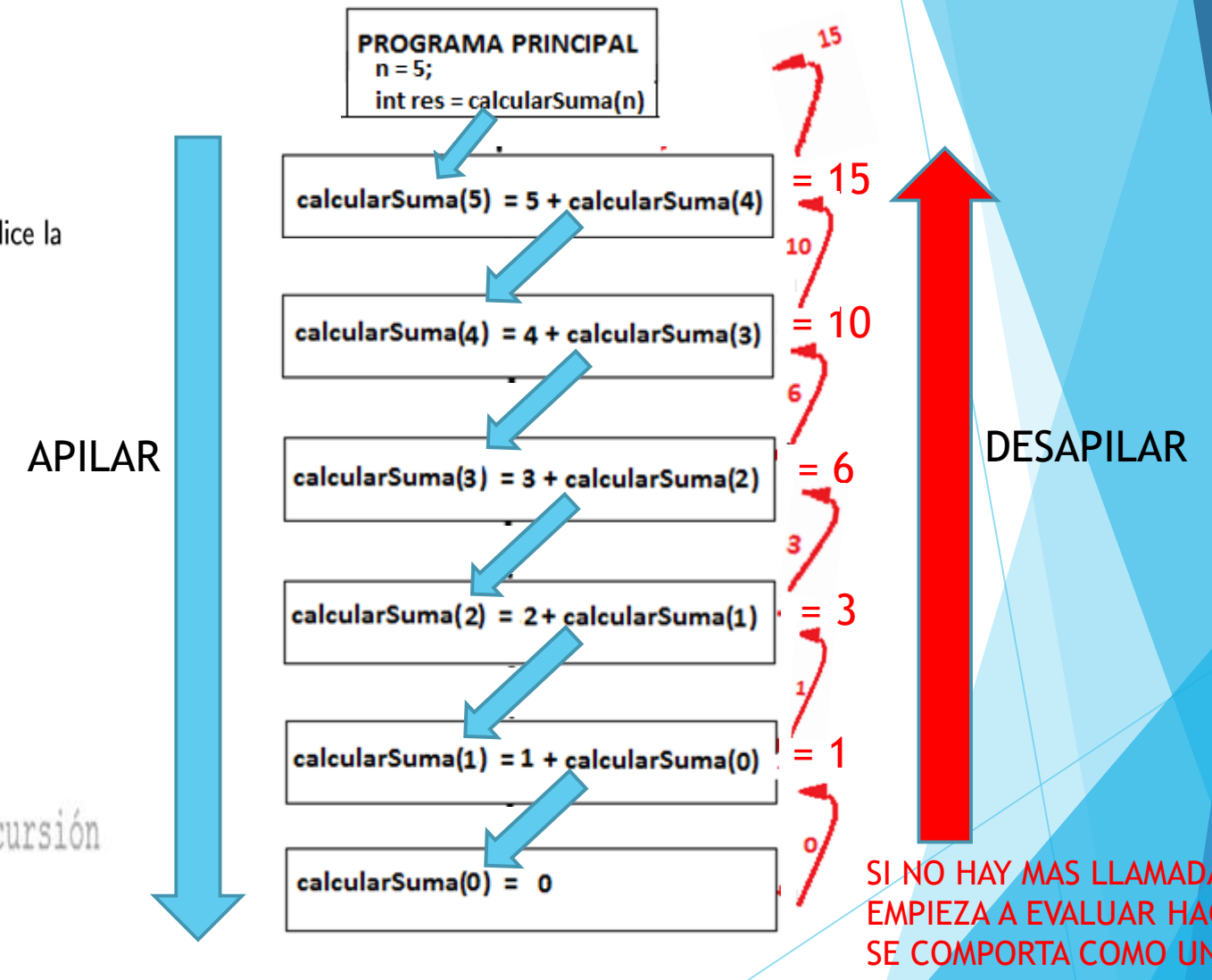
- Cuando un algoritmo se llama a si mismo es como si se ejecutase un nuevo proceso mientras el anterior permanece dormido.
    - Esto se consigue guardando en una pila la dirección de retorno y elementos locales al algoritmo que realiza la llamada:
        - dirección de retorno, constantes, variables, tipos, funciones y procedimientos.
- Es fácil suponer que sin caso base las infinitas llamadas recursivas provocarían un desbordamiento de pila (*Stack Overflow*).
- El código que permite la recursión es añadido automáticamente cuando el compilador detecta una llamada recursiva.

¿Cómo funciona?

En general un algoritmo recursivo se compone de dos partes:

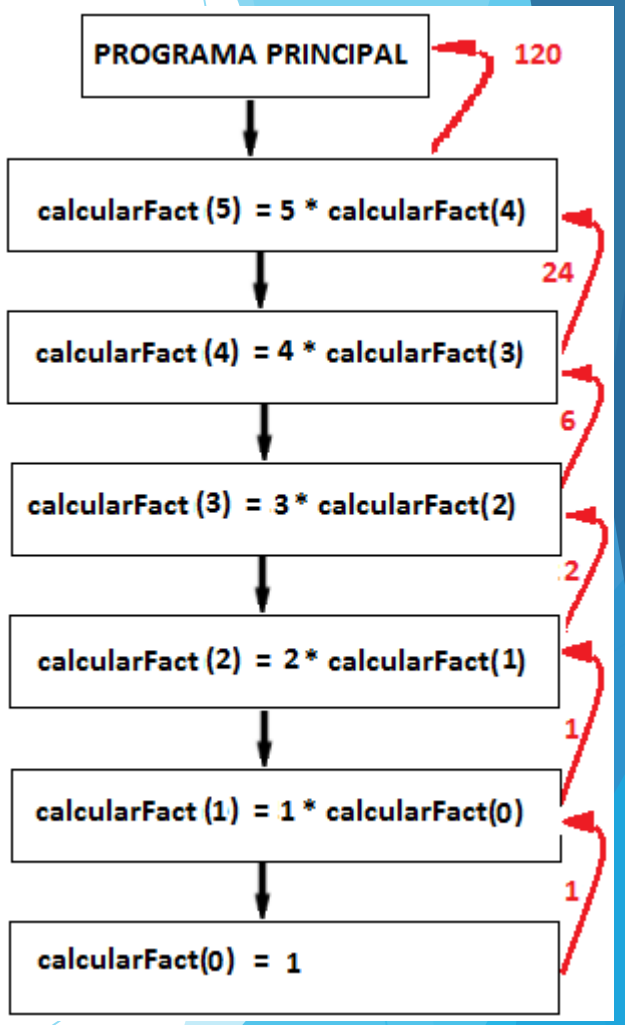
1. **Caso base:** Es la condición de termino de la recursión.
2. **Aplicación de recursión:** Se tratan los datos de tal forma que se realice la llamada recursiva.

```
//SUMAR LOS N PRIMEROS NÚMEROS ENTEROS POSITIVOS
int calcularSuma(int n){
    if (n==0) //Caso base
        return 0;
    else
        return n+calcularSuma(n-1); //Aplicación de recursión
}
```



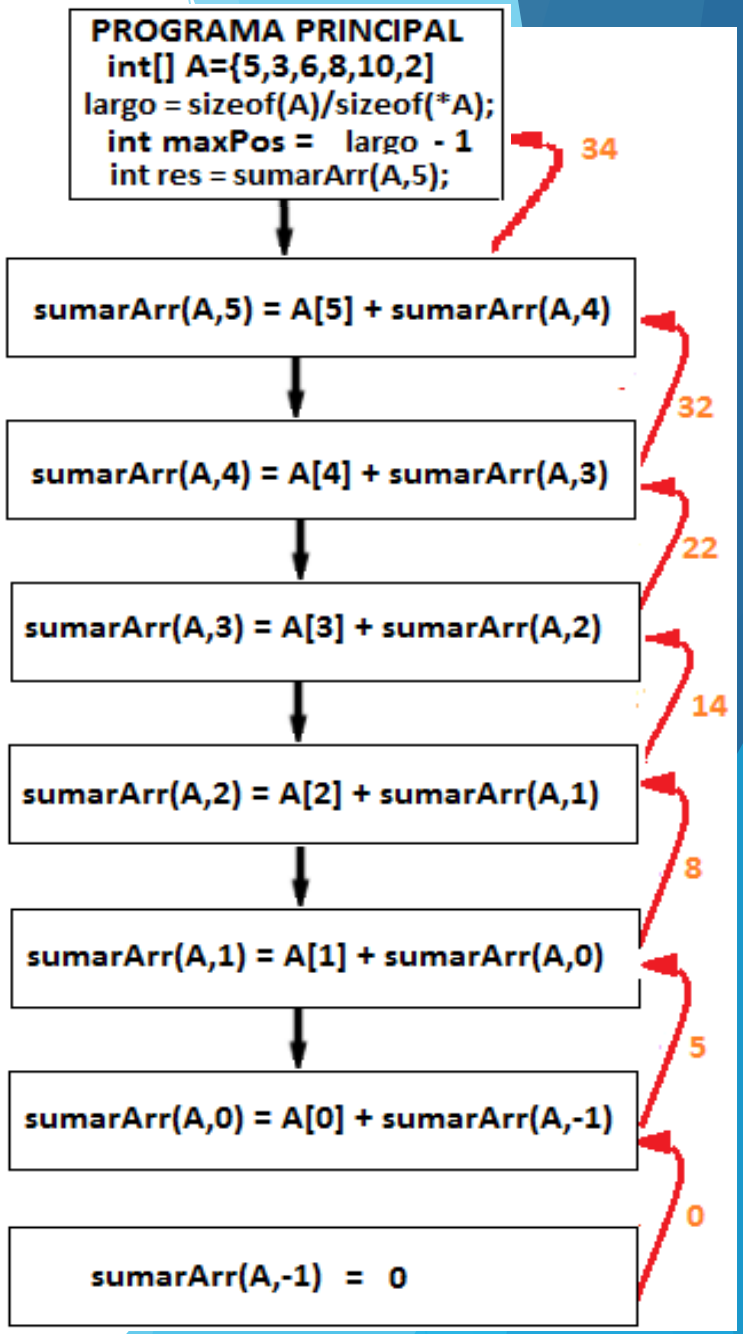
Ejemplo - Calcular el Factorial de un número

```
//FACTORIAL RECURSIVO
int calcularFactorial(int n){
    if (n==0) //Caso base
        return 1;
    else
        return n*calcularFactorial(n-1); //Aplicación de recursión
}
```



Ejemplo – Calcular la suma de los elementos de un arreglo

```
//SUMA ELEMENTOS DE UN ARREGLO RECURSIVO
int sumElemArreglo(int[] A, int i){
    if(i==-1) //Caso base
        return 0;
    else
        return A[i] + sumElemArreglo(A,i-1); //Aplicación de recursión
}
```



Muestre el resultado en ambos casos imprimir arreglo:

```c
int main(){
    int A[] = {5,3,6,8,10,2};
    int largo = sizeof(A)/sizeof(*A);
    printf("El largo es: %d",largo);
    imprimirArreglo(largo-1,A);
    return 0;
}
```

CASO 1:

```c
void imprimirArreglo(int i,int A[]){
    if (i==-1)
        printf("\n");
    else
    {
        imprimirArreglo(i-1,A);
        printf("\nValor posicion %d es: %d",i, A[i]);
    }
}
```

CASO 2:

```c
void imprimirArreglo(int i,int A[]){
    if (i==-1)
        printf("\n");
    else
    {
        printf("\nValor posicion %d es: %d",i, A[i]);
        imprimirArreglo(i-1,A);
    }
}
```

Muestre el resultado en ambos casos recorrer lista:

```c
int main(){
    Lista *L = crearLista();
    insertarElemento(L,1);
    insertarElemento(L,2);
    insertarElemento(L,3);
    recorrerLista(L->ini);
    return 0;
}
```

CASO 1:

```c
void recorrerLista(Nodo *aux){
    if(aux==NULL)
        printf("\n");
    else{
        recorrerLista(aux->sgte);
        printf("\n%d", aux->elem->val);
    }

}
```

CASO 2:

```c
void recorrerLista(Nodo *aux){
    if(aux==NULL)
        printf("\n");
    else{
        printf("\n%d", aux->elem->val);
        recorrerLista(aux->sgte);
    }

}
```

¿Recurrencia o Iteración?

- Para toda función recursiva podemos encontrar otra que haga el mismo cálculo de modo iterativo.
- Pero no siempre es fácil hacer esa conversión.
- En ocasiones, la versión recursiva es más elegante y legible que la iterativa (o, cuando menos, se parece más la definición matemática).
- Las versiones iterativas suelen ser más eficientes que las recursivas, pues cada llamada a una función supone pagar una pequeña penalización en tiempo de cálculo y espacio de memoria, ya que se consume memoria y algo de tiempo en gestionar la pila de llamadas a función.

Entonces, ¿Cuando usar recursividad?:

- Cuando la solución recursiva no es más exigente que la iterativa
- Cuando la solución iterativa es de elevada complejidad

Ejemplo: Sucesión de Fibonacci

La sucesión de Fibonacci es la siguiente sucesión infinita de números naturales.

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377$$

Los números de Fibonacci quedan definidos por la ecuación:

$$f_n = f_{n-1} + f_{n-2}$$

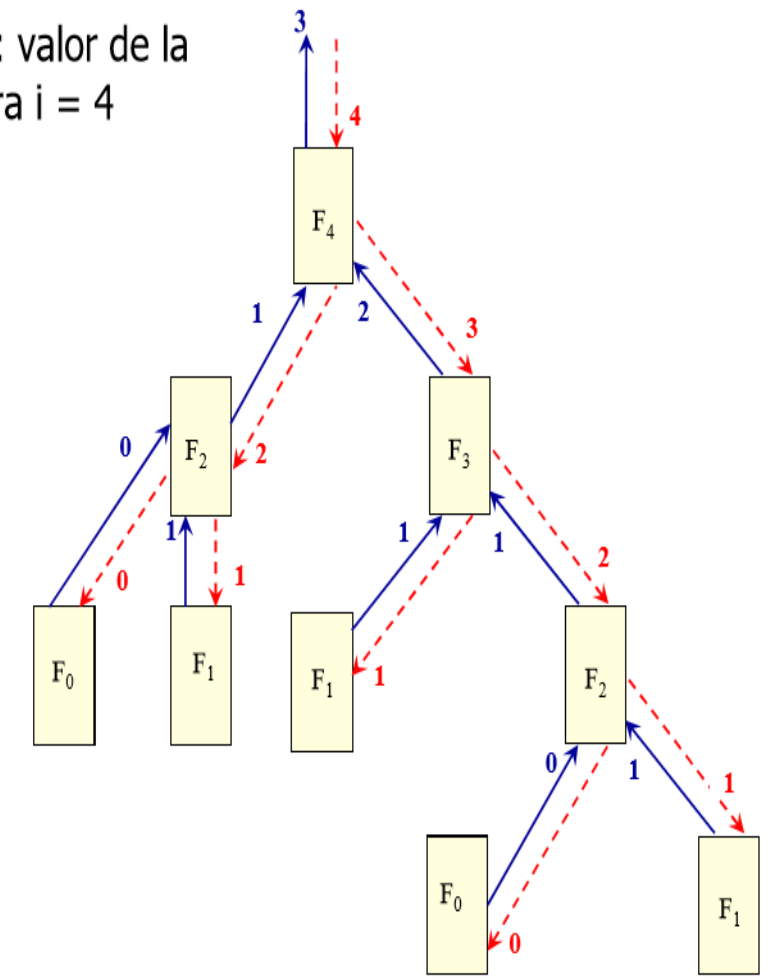
Podemos re-expresar esta definición matemáticamente así:

$$f_n = \begin{cases} 0 & si \quad n = 0 \\ 1 & si \quad n = 1 \\ f_{n-1} + f_{n-2} & si \quad n > 1 \end{cases}$$

- En este caso el número de llamadas es mayor y crece mucho más rápido.

```
1  int fibonacci(int n)
2  {
3    if (n == 0 || n == 1)
4      return n;
5    else
6      return fibonacci(n-1) + fibonacci(n-2);
7  }
```

Ejemplo: valor de la serie para i = 4



Recursión Infinita

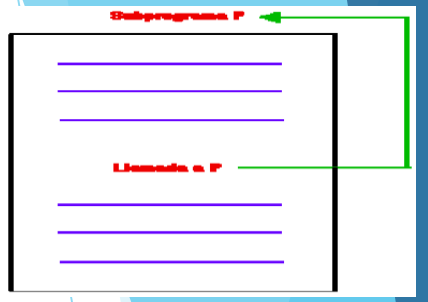
- Una elección inapropiada de los casos base puede conducir a una recursión que no se detiene jamás. Es lo que se conoce por recursión infinita y es análoga a los bucles infinitos.
- Por ejemplo, imagina que deseamos implementar el cálculo recursivo del factorial y diseñamos esta función errónea:

```
1 int factorial(int n)
2 {
3   if(n == 1)
4     return 1;
5   else
6     return n * factorial(n-1);
7 }
```

- ¿Qué ocurre si calculamos con ella el factorial de 0, que es 1?
- Se dispara una cadena infinita de llamadas recursivas, pues el factorial de 0 llama a factorial de -1, que a su vez llama factorial de -2, y así sucesivamente.
- **Jamás llegaremos al caso base**.
- El computador no se quedará colgado indefinidamente: el programa acabará por provocar una excepción. ¿Por qué? Porque la pila de llamadas irá creciendo hasta ocupar toda la memoria disponible, y entonces Python y C indicará que se produjo un "desbordamiento de pila" (en inglés, *stack overflow*).

Recursividad Directa e Indirecta

- **Recursividad directa**: Es la que vimos.
- **Recursividad indirecta**: es cuando, en su definición, llama a otro que, a su vez, contiene al menos una llamada directa o indirecta al primero.



Recursividad indirecta

Número para o impar

```
1  int par(int a)
2  {
3    if(a==0)
4      return 1;
5    else
6      return impar(a-1);
7  }
8
9  int impar(int a)
10 {
11   if(a==0)
12     return 0;
13   else
14     return par(a-1);
15 }
```